Notes on type-driven development with Idris

Christiano Braga

August, 2019

Universidade Federal Fluminense

Introduction

Contact information

Christiano Braga
Associated Professor

Instituto de Computação
Universidade Federal Fluminense
cbraga@ic.uff.br
http://www.ic.uff.br/~cbraga
Lattes Curriculum Vitae

Context i

- Current distributed applications ecosystem: IOT, Cloud, Web...
- A common problem in distributed information systems: *SQL code injection*.
  - Examples: Sony in 2011 and Yahoo! in 2012.
  - Losses of millions of dollars
- The problem, by example:

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = "
        + txtUserId;
```

Context ii

If `txtUserId` is equal to `105 OR 1=1`, which is always true, a malicious user may access *all* user information from a database.

- Typical solutions
  - SQL parameters: additional values are passed to the query.
  - Escaping functions: they transform the input string into a "safe" one before sending it to the DBMS.
- The problem with the solutions is that communication relies on *strings*.
- What if we could **type** this information?
- Web programming invariably requires following certain **protocols**.

Context iii

- For example, to connect to make a query:
  1. Create a connection.
  2. Make sure the connection was established.
  3. Prepare an SQL statement.
  4. Make sure that variables are bound.
  5. Execute the query.
  6. Process the result of the query.
  7. Close connection.

- Of course, a function could implement such a sequence, but how could one make sure that such a sequence is *always* followed?
- In other words, what if we could *type* protocol behavior and make sure our Web programs *cope* with such types?

Context iv

- Moreover, what if we could define special *notation* to create instances of such types?
- Prtocols are one example but note that *business processes* may be treated the same way.

Service-oriented web development model

Services are blackboxes, are stateless, are composable, among other nice characteristics.

- Services are first-class citizens in Cloud PaaS, and other platforms.
- These characteristics allow for a *clean* and *simple* interpretation of services as *functions*.
- ***What about capturing a company's way of developing PaaS as DSL?***
- ***What about capturing a company's clients processes as DSL?***

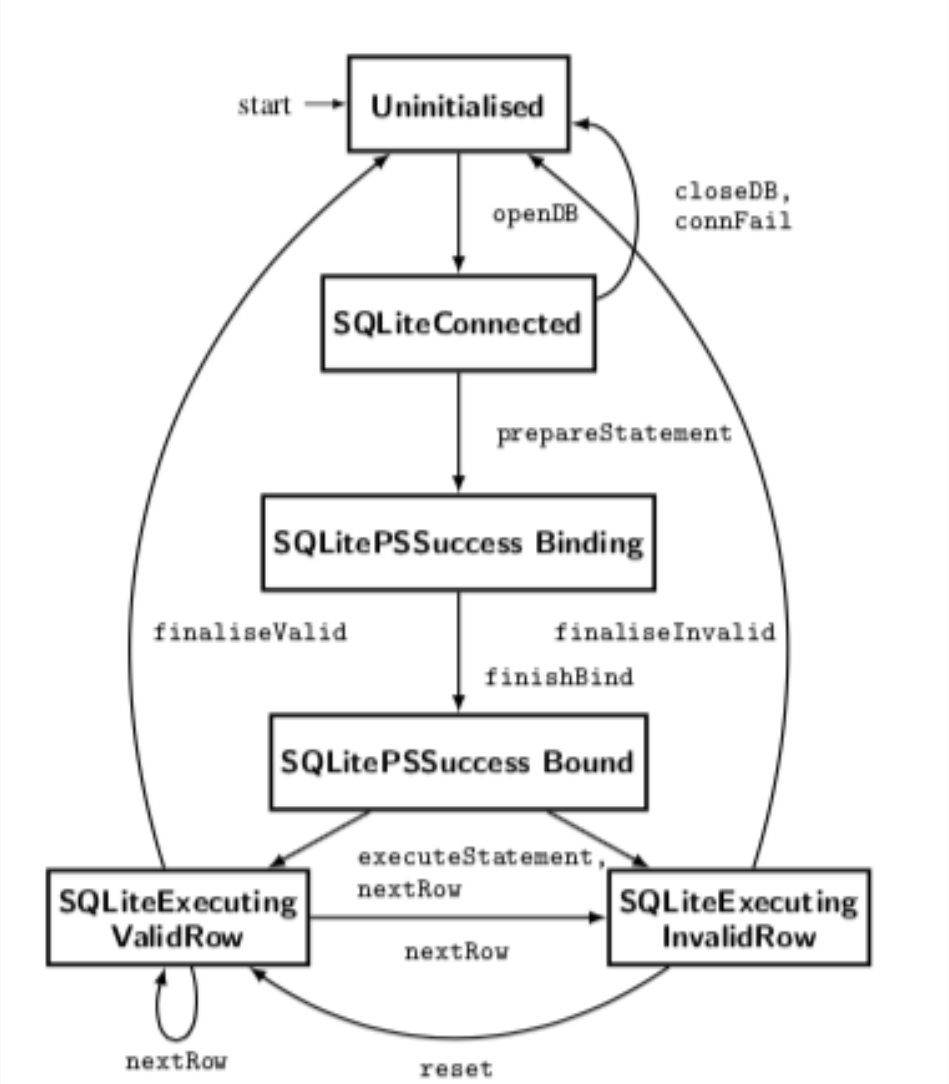
An example DSL

(From Fowler&Brady13.)

- Think of each step of a Web application as a business process.
- The notion of a Web application is typed, and so are its steps.
- For example, a Web application has forms and its forms have handlers.
- A particular Web application is *safe* (or well-typed) if its forms are well-typed. A form is well-typed if its handlers are also well-typed.

An example DSL ii

- The database protocol can be captured as a type.



An example DSL iii

- For example, the step `SQLiteConnected` step has type

```
data SQLiteConnected : Type where
     SQLConnection : ConnectionPtr -> SQLiteConnected
```

- The DSL has constructions for defining typed form handlers such as

```
handleRequest : CGIProg
     [SESSION (SessionRes SessionUninitialised),
      SQLITE ()] ()
```

that will only handle a request on properly established sessions.

Programming languages support for DSL development

- Essentially, there are two approaches for DSL-based development:
  1. Transformational approach:
     DSL program ***=parsing=>*** Protocol data type instance ***=transformation=>*** Web (micro)service framework.
  2. Embedded DSL approach:
     The programming languages has support the definition of notation and typing.
- Programming languages that support approach #i are Racket and Maude.
- Programming languages that support approach #ii are Idris, Lean and Haskell.

Our research approach

- To program services with domain-specific languages, implemented on top of strongly typed functional languages.
- To develop and apply program analysis techniques to DSL-based approaches to Web development.
- More specifically, to develop Web programming support in Idris.

Summing up

- We have chosen an important technical problem in web development (SQL injection), that may cause loss of millions of dollars, to illustrate DSL with functional programming usefulness.
- The issues raised here may be moved to a higher level of abstraction to represent buiness processes and their refinement into code.
- There is off the shelf technology to support this approach.

Suggested reading

Simon Fowler and Edwin Brady. 2013. Dependent Types for Safe and Secure Web Programming. In Proceedings of the 25th symposium on Implementation and Application of Functional Languages (IFL '13). ACM, New York, NY, USA, Pages 49, 12 pages. DOI: https://doi.org/10.1145/2620678.2620683

The need for types

The need for types

This section motivates the use of strong typing with a very very simple example: Bhaskara's theorem. In a tutorial way, we illustrate how types are necessary and, more specifically, how Idris' strong-typing presents itslef as a powerful development tool.

Bhaskara's theorem

From school: Bhaskara's theorem[1]

$$ax^2 + bx + c = 0 \quad \Rightarrow \quad x = \frac{-b \overset{+}{-} \sqrt{\delta}}{2a}$$
$$\text{where } \delta = b^2 - 4acb$$

As functions

$$\texttt{bhask}(a, b, c) =$$
$$\left(-b + \sqrt{\texttt{delta}(a, b, c)}/2a, -b - \sqrt{\texttt{delta}(a, b, c)}/2a\right)$$

$$\texttt{delta}(a, b, c) = \quad b^2 - 4acb$$

First attempt: no types. i

- In Python:

```
from math import sqrt

def delta(a,b,c):
    return  (b * b) - (4 * a * c)

def bhask(a,b,c):
    d = delta(a,b,c)
    sr = sqrt(d)
    r1 = (-b + sr) / 2 * a
    r2 = (-b - sr) / 2 * a
    return (r1, r2)
```

First attempt: no types. ii

- When we run `bhask(1,2,3)` the following is spit out:

```
Traceback (most recent call last):
  File "bhask.py", line 16, in <module>
    bhask(1,2,3)
  File "bhask.py", line 9, in bhask
    sr = sqrt(d)
ValueError: math domain error
```

- This cryptic answer is only because we rushed into a direct implementation and forgot that `delta(a,b,c)` may return a *negative* value!

Second attempt: still no types. i

- Now, assuming we are instered only on Real results, how should `bhask` deal with the possilibity of a negative `delta`?
- One possibilty is to raise an *exception*:

```
from math import sqrt

def delta(a,b,c):
    return  (b * b) - (4 * a * c)

def bhask(a,b,c):
    d = delta(a,b,c)
    if d >= 0:
        sr = sqrt(d)
```

Second attempt: still no types. ii

```
            r1 = (-b + sr) / 2 * a
            r2 = (-b - sr) / 2 * a
            return (r1, r2)
        else:
            raise Exception("No Real results.")
```

- This implementation gives us a more *precise* answer:

```
Tue Jul 30@17:18:02:sc$ python3 -i bhask.py
Traceback (most recent call last):
  File "bhask.py", line 16, in <module>
    bhask(1,2,3)
  File "bhask.py", line 14, in bhask
```

Second attempt: still no types. iii

```
    raise Exception("No Real results.")
Exception: No Real results.
```

- A very **important** point here is that we only find all this out while actually *running* our implementation. Can't we do better? That is, let the **compiler** find out that `delta` may become a negative number and complain if this is not properly handled?

Third attempt: Idris. i

- Let us play with `delta` first.
- Strongly-typed languages, such as Idris, force us to think about types right away as we need to define `delta`'s signature. If we make the same mistake we did in the first attempt and forget that `delta` may become negative, we may write,

```
> delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Nat
> delta a b c = (b * b) - (4 * a * c)
```

the compiler would tell us:

Third attempt: Idris. ii

```
Type checking ./intro.lidr
intro.lidr:100:26:
    |
100 | > delta a b c =  (b * b) - (4 * a * c)
    |                          ^
When checking right hand side of delta with expected type
        Nat

When checking argument smaller to function Prelude.Nat.-:
     Can't find a value of type
     LTE (mult (plus a (plus a (plus a (plus a 0)))) c)
     (mult b b)
```

Third attempt: Idris. iii

- This is cryptic, in a first-glance, but tells us precisely **what** is wrong **and** at **compile** time. The problem is **with subtraction**: the type checker was not able to solve the inequality, defined in Idris' libraries,

$$4ac \leq b^2$$

in order to produce a **natural** number while computing `delta`, as natural numbers can not be negative!

First fix. i

- And we have not even started thinking about `bhask` yet! But let us first make `delta` type right by changing its signature:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (b * b) - (4 * a * c)
```

- To see the effect of this change, load `delta-fix.lidr` with the command:

```
:l delta-fix.lidr
```

- Don't be so happy though! This is not what we want yet.

First fix. ii

```
Type checking ./delta-fix.lidr
delta-fix.lidr:5:18-38:
  |
5 | > delta a b c =  (b * b) - (4 * a * c)
  |                  ~~~~~~~~~~~~~~~~~~~~~
When checking right hand side of delta with expected type
        Int

Can't disambiguate since no name has a suitable type:
        Prelude.Interfaces.-, Prelude.Nat.-

Holes: Main.delta
```

First fix. iii

- Idris does not know which subtraction operation to use because we are operating operating with natural numbers but we should return an integer! A casting is in order!

Second fix. i

- Think about why we should cast the right-hand side expression in the following way:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (cast (b * b)) - (cast (4 * a * c))
```

and not the whole right-hand side of `delta` at once. - To see the effect of this change, load `delta-fix2.lidr` with the command:

```
:l delta-fix2.lidr
```

- You should finally be able to see

```
Type checking ./delta-fix2.lidr
*delta-fix2>
```

Second fix. ii

and run `delta 1 2 3`, for instance, to see the following result.

```
*delta-fix2> delta 1 2 3
-8 : Int
```

- Your session should look like this at this point:

```
Mon Aug 05@14:24:16:the-need-for-types$ idris --nobanner tn
Type checking ./tnft.lidr
tnft.lidr:107:25:
    |
107 | > delta a b c = (b * b) - (4 * a * c)
    |                         ^
When checking right hand side of delta with expected type
        Nat
```

Second fix. iii

```
When checking argument smaller to function Prelude.Nat.-:
     Can't find a value of type
             LTE (mult (plus a (plus a (plus a (plus a 0))))
                 (mult b b)

Holes: Main.delta
*tnft> :l delta-fix.lidr
Type checking ./delta-fix.lidr
delta-fix.lidr:5:18-38:
  |
5 | > delta a b c =  (b * b) - (4 * a * c)
  |                  ~~~~~~~~~~~~~~~~~~~~~
```

Second fix. iv

```
When checking right hand side of delta with expected type
        Int

Can't disambiguate since no name has a suitable type:
        Prelude.Interfaces.-, Prelude.Nat.-

Holes: Main.delta
*delta-fix> :l delta-fix2.lidr
*delta-fix2> delta 1 2 3
-8 : Int
```

Bhaskara at last! i

- Painful, no?
  No!
- The compiler is our *friend* and true friends do not always bring us good news!
- Think about it using this metaphor: do you prefer a shallow friend, such as Python, that says yes to (almost) everything we say (at compile time), but is not there for us when we really need it (at run time), or a *true* friend, such as Idris, that tells us that things are not all right all the time, but is there for us when we need it?

Bhaskara at last! ii

- Another way to put it is that "With great power comes great responsibility!", as the philosopher Ben Parker used to say... Strong typing, and in particular this form of strong typing, that relies on *automated theorem proving* requires some effort from our part in order to precisely tell the compiler how things should be.
- Having said that, let us finish this example by writing `bhask` function.

Bhaskara: first attempt i

- Bhaskara's solution for second-degree polynomials gives no Real solution (when $\delta < 0$), one (when $\delta = 0$), or two (when $\delta > 0$). Since "The Winter is Coming" we should be prepared for two roots:

```
bhask : (a :  Nat) -> (b : Nat) -> (c : Nat) -> (Double, Do
bhask a b c = ((-b + (sqrt (delta a b c))) / (2 * a),
               (-b - (sqrt (delta a b c))) / (2 * a))
```

- Moreover, we should now work with the Idris `Double` type, because of the `sqrt` function. Run

```
*bhask-fun> :t sqrt
sqrt : Double -> Double
```

Bhaskara: first attempt ii

- Again, our naivete plays a trick on us:

```
Type checking ./bhask-fun.lidr
bhask-fun.lidr:2:19:
  |
2 | > bhask a b c = ((-b + (sqrt (delta a b c))) / (2 * a),
                    (-b - (sqrt (delta a b c))) / (2 * a))
  |                   ^
When checking right hand side of bhask with expected type
        (Double, Double)

When checking an application of function Prelude.Interfaces
        Type mismatch between
                Nat (Type of b)
```

Bhaskara: first attempt iii

```
and
                Double (Expected type)
```

Load file `bhask-fun.lidr` to see this effect.

- We should write `negate b` instead of `- b`, as `-` is a *binary* operation only in Idris. Moreover, we should *not* be able to negate a natural number! Again, casting is necessary.

Bhaskara: final attempt i

- Let us fix all casting problem at once, the final definitions should be as follows:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (cast (b * b)) - (cast (4 * a * c))
bhask : (a :  Nat) -> (b : Nat) -> (c : Nat) -> (Double, Do
bhask a b c =
  (negate (cast b) + (sqrt (cast (delta a b c))) / cast (2
   negate (cast b) - (sqrt (cast (delta a b c))) / cast (2
```

- We can now play with `bhask`, after executing `:l bhask-fun-fix.lidr`

Bhaskara: final attempt ii

```
Type checking ./bhask-fun-fix.lidr
*bhask-fun-fix> bhask 1 10 4
(-5.41742430504416, -14.582575694955839) : (Double, Double)
*bhask-fun-fix> bhask 1 2 3
(NaN, NaN) : (Double, Double)
```

- Note that when $\delta < 0$ Idris gives a `NaN` value, which stands for *Not a number*. In other words, `bhask` is **total** as opposed to the **partial** approach in Python where we needed to raise an exception to capture the situation where the roots are not Real numbers.
- Idris can help us identify when a function is total. We simply need to run:

Bhaskara: final attempt iii

```
*bhask-fun-fix> :total bhask
Main.bhask is Total
```

Wrapping-up i

- First and foremost motivate strong-typing in Idris.
- Introduce notation for functions in Idris. The signature of a function, such as `delta` includes a name, formal parameters and a return type, such as:
  `delta : (a : Nat) -> (b : Nat) -> (c : Nat) -> Int`.
- The formal parameters of a function are declared using the so-called Currying form (after Haskell Curry): currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Wrapping-up ii

- This allows to *partially apply* a function! For instance, we can call `delta 1 2`. This will produce a function that expects a number and then behaves as `delta`.
- Take a look at the following session:

```
*bhask-fun-fix> delta
delta : Nat -> Nat -> Nat -> Int
*bhask-fun-fix> delta 1
delta 1 : Nat -> Nat -> Int
*bhask-fun-fix> delta 1 2
delta 1 2 : Nat -> Int
*bhask-fun-fix> delta 1 2 3
-8 : Int
```

Wrapping-up iii

```
*bhask-fun-fix> (delta 1) 2
delta 1 2 : Nat -> Int
*bhask-fun-fix> ((delta 1) 2) 3
-8 : Int
```

- At the end of the day, `delta 1 2 3` is just *syntax sugar* for `((delta 1) 2) 3`.

Wrapping-up iv

- Total functions. From Idris' FAQ: Idris can't decide in general whether a program is terminating due to the undecidability of the Halting Problem. It is possible, however, to identify some programs which are definitely terminating. Idris does this using "size change termination" which looks for recursive paths from a function back to itself. On such a path, there must be at least one argument which converges to a base case.
- Mutually recursive functions are supported. However, all functions on the path must be fully applied. In particular, higher order applications are not supported

Wrapping-up v

- Idris identifies arguments which converge to a base case by looking for recursive calls to syntactically smaller arguments of inputs. e.g. k is syntactically smaller than $S(Sk)$ because k is a subterm of $S(Sk)$, but (k, k) is not syntactically smaller than (Sk, Sk).
- Type casting. We have used `cast` many times in order to *inject* our values from one type into another.
- Some Read-Eval-Print-Loop (REPL) commands. We have seen how to load a file with `:l`, check its type with `:t`, and check weather a function is total or not with `:total`.

Type-define-refine approach

Type-define-refine approach

The approach is threefold:

1. Type—Either write a type to begin the process, or inspect the type of a hole to decide how to continue the process.
2. Define—Create the structure of a function definition either by creating an outline of a definition or breaking it down into smaller components.
3. Refine—Improve an existing definition either by filling in a hole or making its type more precise.

Following the TDD book, we use the Atom editor to illustrate the process. (Idris defines an IDE API such that editors like Atom, Emacs or Vi can interact with the REPL.)

The `allLenghts` function i

Let us write a function that given a list of strings computes a list of integers denoting the lenght of each string in the given list.

- Type

Which should be the type for `allLengths`? Our "problem statement" has alredy specified it so we just have to write it down:

```
allLenghts : List String -> List Nat
```

After loading the file `tdr.lidr` we get the following.

The allLenghts function ii

```
Type checking ./tdr.lidr
Holes: Main.allLenghts
*tdr> allLenghts
allLenghts : List String -> List Nat
Holes: Main.allLenghts
```

There is no surprise with the type but there is Hole in our program. Obviously is because we did not declare the equations that define allLenghts. This may also occur when Idris fails to type-check a given program.

- Define

The `allLenghts` function iii

Idris may help us think about which cases our function must handle. In the Atom editor, we press Ctrl+Alt+A, producing the following definition:

```
allLenghts : List String -> List Nat
allLenghts xs = ?allLenghts_rhs
```

Of course this is not enough. Here is what Idris says when we load it like this:

```
Type checking ./tdr.lidr
Holes: Main.allLenghts_rhs
```

The `allLenghts` function iv

Let us think about it: what just happened here? Nothing more than create an equation saying that when the `xs` list is given, "something" `?allLenghts_rhs`-ish will happen. Simple but useful when we repeat this process. It is even more useful as a learning tool. Let's continue!

Idris won't leave us with our hands hanging here. It can assist us on thinking about what `?allLenghts_rhs` should look like if we inspect `xs`.

If we press Ctrl+Alt+C on `xs` the editor spits out the following code:

The `allLenghts` function v

```
allLenghts : List String -> List Nat
allLenghts [] = ?allLenghts_rhs_1
allLenghts (x :: xs) = ?allLenghts_rhs_2
```

Two equations were produced because lists in Idris are defined either as the emoty list, denoted by `[]`, or a non-empty list denoted by the *pattern* `x :: xs`, where `x` is the first element of the given list, whcih is concatenated to the rest of list in `xs` by the operator `::`.

Nice, and now we have two holes to think about, when the given list is empty and otherwise. Idris allows us to check the type of each hole using the comand Ctrl+Alt+T when the cursor is on top of each variable.

The `allLenghts` function vi

```
--------------------------------------
allLenghts_rhs_1 : List Nat

x : String
xs : List String
--------------------------------------
allLenghts_rhs_2 : List Nat
```

- Refine

The `allLenghts` function vii

The refinement of `allLenghts_rhs_1` is trivial: Ctrl+Alt+S (*proof search*) on it gives us `[]`.

For `allLenghts_rhs_2` we need to know however that there exists a `length` operation on strings. We should than apply it `x` and "magically" build the rest of the resulting string. Our code now looks like this:

```
allLenghts : List String -> List Nat
allLenghts [] = []
allLenghts (x :: xs) = (length x) :: ?magic
```

Atom and Idris may help us identify what kind of magic is this. We just have to Ctrl+Alt+T it to get:

The `allLenghts` function viii

```
x : String
xs : List String
------------------------------------
magic : List Nat
```

So now we need *faith on recursion* (as Roberto Ierusalimschy, a co-author of Lua, says) and let the rest of the problem "solve itself". Finally, we reach the following implementation:

```
> module Main
>
> allLengths : List String -> List Nat
```

The allLenghts function ix

```
> allLengths [] = []
> allLengths (x :: xs) = (length x) :: allLengths xs
```

Awesome! For our final magic trick, I would like to know if Idris has a function that given a string produces a list of strings whose elements are the substrings of the first. Try this on the REPL:

```
*type-define-refine/tdr> :search String -> List String
= Prelude.Strings.lines : String -> List String
Splits a string into a list of newline separated strings.

= Prelude.Strings.words : String -> List String
Splits a string into a list of whitespace separated strings
...
```

The `allLenghts` function x

It turns out that `words` is exactly what I was looking for! Run the following:

```
*type-define-refine/tdr> :let l = "Here we are, born to be
 we are princess of the universe!"
*type-define-refine/tdr> words l
["Here",
 "we",
 "are,",
 "born",
 "to",
 "be",
 "kings,",
 "we",
```

The allLenghts function xi

```
  "are",
  "princess",
  "of",
  "the",
  "universe!"] : List String
```

And Finally

```
*type-define-refine/tdr> :let w = words l
*type-define-refine/tdr> allLenghts w
[4, 2, 4, 4, 2, 2, 6, 2, 3, 8, 2, 3, 9] : List Nat
```

Lab i

In the labs in this short-course you will have to complete or fix some Idris code.

- First lab.

The first lab is to complete the code below using what we have discussed so far.

```
> wordCount : String -> Nat
> -- Type-define-refine this function!
> -- Start by running `Ctrl+Alt+A` to add a definition, th
> -- to split cases and finally `Ctrl+Alt+S` to search for
> -- the code you need! (Intrigued? Ask the instructor for
```

Lab ii

```
> -- this topic than = )
>
> average : (str : String) -> Double
> average str = let numWords = wordCount str
>                   totalLength = sum (allLengths (words st
>               in ?w
> -- Which is the type of `?w1`?
> -- Proof search won't help you here, unfortunately...
> -- Run `:doc sum` at the REPL. Just read the documentatio
> -- not the type of `sum`.
>
> showAverage : String -> String
> showAverage str =
```

Lab iii

```
>    let m = "The average word length is: "
>        a = average ?w
>    in m ++ show (a) ++ "\n"
> -- Check the type o `w` and think about it!
>
> main : IO ()
> main = repl "Enter a string: " showAverage
```

- Using the example string from above, you should get the following spit at you:

Lab iv

```
Sat Aug 03@18:05:17:type-define-refine$ idris --nobanner t
Type checking ./tdr.lidr
*tdr> :exec main
Enter a string: Here we are, born to be kings, we are princ
The average word length is: 3.923076923076923
```

- Moreover, you may *compile it* to an executable with the following command line:

```
idris --nobanner tdr.lidr -o tdr
```

and then execute it, as follows.

```
Sun Aug 04@12:39:21:type-define-refine$ ./tdr
Enter a string:
```

The need for dependent types

The need for dependent types

Overflow conditions in software appear to be a simple thing to implement. An important counter-example is the Ariane 5 rocket that exploded due to a down cast from 64-bit number into a 16-bit one.

> *The Ariane 5 had cost nearly $8 billion to develop, and was carrying a $500 million satellite payload when it exploded.*

11 of the most costly software errors in history

In this chapter we look at a simpliflied version of the `Vector` datatype, avaiable in Idris' library, to try and understand how *dependent typing* can be useful to have type-safe array handling that could help prevent catastrophes such as the Ariane 5 explosion.

Vector

A datatype is nothing but an implementation of some "domain of information". It could very well represent low level information such as data acquired by a sensor in a Internet of Things (IoT) system or the structure that organizes the decision making process in planning.

Our datatype here is quite simple but illustrates very well how dependent types may help safe data modeling and implementation.

```
> module Vect
> data Vect : Nat -> Type -> Type where
>     Nil  : Vect Z a
>     (::) : (x : a) -> (xs : Vect k a) -> Vect (S k) a
```

An array or vector is built or *constructed* using either one of the constructor operations (unary) `Nil` or (binary) `::`. (The `module` keyword here simply defines a *namespace* where `Vect` will live.)

Dependency

Let's look at the instance first and then to the type declaration. Note that the type of `[1]` is `Vect 1 Integer`. The type of a Vect *depends* on its *size*! Think about examples of vectors in programming languages you know. If you query for the type of a given vector, if at all possible, what the run-time of your programming language will answer?

In Python, for instance, you would get something like,

```
v = [1,2,3]
type(v)
<class 'list'>
```

that is, is a `list` and that's all! In C an array is a pointer! (A reference to a memory address, for crying out loud!)

In Idris, we know it is a vector and its size, an important property of

Wrapping-up

1. Defining datatypes.
2. Defining dependent datatypes.
3. Using dependent datatypes to find errors at compile time.
4. Type expressions.

Insertion sort lab.

Insertion sort lab.

Here is what we will implement: - Given an empty vector, return an empty vector. - Given the head and tail of a vector, *sort* the tail of the vector and then insert the head into the sorted tail such that the result remains sorted.

1. Type We will use the `Vect` datatype avaiable in Idris' prelude.

```
> import Data.Vect
```

And it is easy to grasp the signature of our function, so here it goes.

```
> insSort : Vect n elem -> Vect n elem
```

1. Define Now we add a clause using `Ctrl+Alt+A` on `inSort`, resulting in


```
insSort : Vect n elem -> Vect n elem
```

Programming with type-level functions

Programming with type-level functions

Here are a couple of examples where first-class types can be useful: - Given an HTML form on a web page, you can calculate the type of a function to process inputs in the form. - Given a database schema, you can calculate types for queries on that database. In other words, the type of a value returned by a database query may vary *depending on* the database *schema* and the *query* itself, calculated by **type-level functions**.

This should be useful in a number of contexts such as Data validation in Robotic Process Automation, SQL Injection, (Business) Process Protocol Validation, just to name a few.

In this section we discuss and illustrate how this way of programming is available in the Idris language.

Formatted output example

This examples explores some of the components for the RPA scenario. It exemplifies how to make strings from properly-typed data using type-functions, similarly to the `printf` function in the C programming language.

```
> module Format
>
> data Format =
> Number Format
> | Str Format
> | Lit String Format
> | End
```

The `Format` datatype is an *inductive* one: is a "list" such that its elements are either `Number`, `Str`, `Lit s` (where `s` is string) or `End`.

Caveats

(From TDD book.)

In general, it's best to consider type-level functions in exactly the same way as ordinary functions. This isn't always the case, though. There are a couple of technical differences that are useful to know about:

- Type-level functions exist at *compile* time only. There's no runtime representation of Type, and no way to inspect a Type directly, such as pattern matching.
- Only functions that are total will be evaluated at the type level. A function that isn't total may not terminate, or may not cover all possible inputs. Therefore, to ensure that type-checking itself terminates, functions that are not total are treated as constants at the type level, and don't evaluate further.